Service Programs

By MARSHALL E. BARTON, NEIL M. HALLER, and GUY W. RICKER

(Manuscript received March 6, 1969)

The programming of a large switching machine requires that many service functions be performed on a general purpose computer. This article describes the service program package that supports the No. 2 Electronic Switching System. This package includes an advanced macro assembler to convert symbolic source programs into binary object programs, a loader to combine independent assemblies, and a simulator to provide checkout facilities.

I. INTRODUCTION

The No. 2 Electronic Switching System is a stored program control telephone switching system. The minimum program consists of about 75,000 instructions arranged in 22-bit words and written in more than 50 separate sections. It is clearly impractical to write such a program without extensive computer aids.

1.1 *Functions*

Major steps provided by the No. 2 ESS service programs are shown in Fig. 1. The assembler separately converts each symbolic source program from the language written by the programmer to the binary language of the No. 2 ESS. The loader then combines these separately assembled sections and resolves references between them. The output of the loader may be used by the magnet program to prepare the punched cards used in magnetizing the actual program store, or by the simulator. The simulator, using tables prepared by the assembler, furnishes more powerful program checkout facilities than would be practical on a switching machine and provides these facilities before the laboratory model is available.

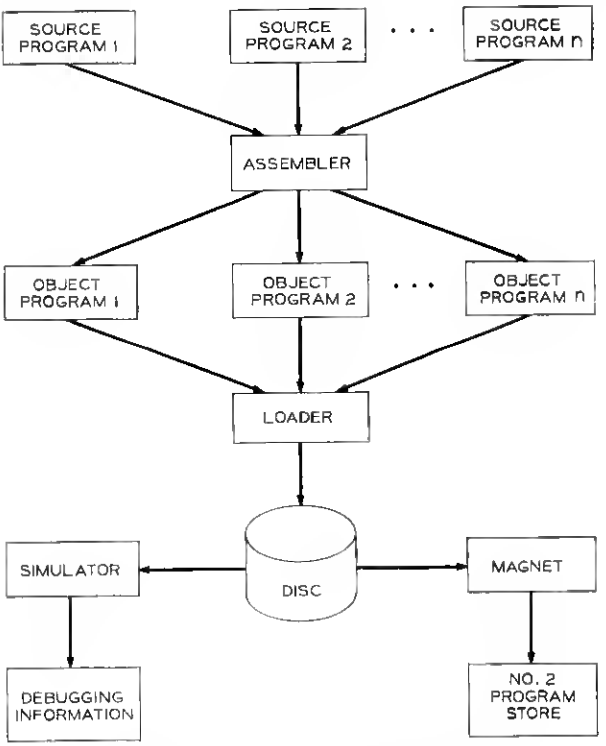


Fig. 1 — Major steps in program processing.

1.2 *Organization*

The No. 2 ESS service programs run on an IBM 360 computer system with at least 262,144 bytes of storage. They operate as a subsystem under Operating System 360. This gives the users a uniform interface to all service programs and isolates them from the complexities of System 360 job control language.

II. ASSEMBLER

The primary purpose of an assembler is to convert a source program written in symbolic language to an object program which is in machine (binary) language. The switching assembler program (SWAP) was designed to do this for No. 2 ESS and several other stored program systems. Each machine instruction is represented by a symbolic code which the assembler translates to the appropriate bit pattern in the object program.

A programmer may assign a symholic representation for any location in the object machine memory and then refer to that location symholically in an instruction. By this method of referencc, the program can be changed without regard for location changes caused by the insertion or deletion of words.

Symbolic addressing also allows a data constant to be defined with a symbolic name and then used in many places. If the value of the constant is changed, all the places it is used need not be hunted and changed. The method of defining data as well as many other functions is done through assembler control instructions called pseudo-operations. A pseudo-operation does not usually result in object code, but tells tbe assembler how to translate symbolic information it will encounter in the program.

2.1 *Input Syntax*

2.1.1 *Fields*

The general form of a SWAP input line is anchored free field. This means that, beyond column one, there are no restrictions as to where on the line the information appears. The four major divisions of an input line are called fields: the location field, if present, must start in column one; the operation field follows the location field; the variable field follows the operation field; and the comment field is last.

Fields must be separated by one or more blanks or a single comma, except for tbe comment field which must start with a sharp sign (#). (If a sharp sign appears in column one, the entire line is treated as commentary.) The variable field, sometimes called the address field, may be terminated in three ways: (*i*) any number of blanks followed hy a sharp, (*ii*) the physical end of the line, or (*iii*) the logical end of line which is indicated by a semicolon. When a line is truncated by a semicolon, the character immediately following the semicolon is considered to be column one of the next line.

2.1.2 *Continuations*

A continuation may bc indicated by the at sign (@) used in either of two places. If an at sign is the last nonblank character on a line, then tbe next line is a continuation. If an at sign appears in column one of a line, then the line with the at sign is a continuation of the previous line. In both cases, the lines are joined at the at sign which is discarded by the assembler.

2.2 *Basic Pseudo-Operations*

2.2.1 EQU, SET, *and* TEXT *Pseudo-Operations*

The most basic pseudo-operations are those that define a symbolic representation for a quantity: the EQU and SET pseudo-operations assign a numeric value to a symbol; the TEXT pseudo-operation defines a symbolic name for a string of characters. While symbols defined with SET and TEXT may be redefined, those defined by EQU may not.

Either of the following lines would assign the value 10 to the symbol SYMB:

```
SYMB   EQU   10
SYMB   SET   10.
```

To define or redefine the symbol VOWELS with the string of five vowels as its value, the following statement would be used:

```
VOWELS TEXT 'AEIOU'
```

2.2.2 JUMP *and* DO *Pseudo-Operations*

The pseudo-operations that allow the normal sequence of processing to be modified provide the real power of an assembler. In SWAP, the pseudo-operations that provide that control are JUMP and DO. JUMP forces the assembler to continue sequential processing with the indicated line, ignoring any intervening lines. As an example, consider the following sequence of lines:

```
        .
        .
        .
JUMP    .LINE
A   EQU   2
.LINE;B EQU   3
        .
        .
        .
```

The symbol A will not be defined because that line would be skipped under control of the JUMP. The symbol .LINE is called a sequence symbol and is treated not as a normal location field but only as the destination of a JUMP. The first character of a sequence symbol must be a period. The line that is "jumped" to may be either before or after the JUMP statement.

The JUMP is taken conditionally when an expression is used as the following example shows:

```
INC   SET   0
.AA;INC   SET   INC+1          # INCREASE COUNTER
   JUMP   .XX,INC>10           # IS IT OVER LIMIT
    .
    .
    .
   JUMP   .AA                  # GO AROUND AGAIN
.XX
```

The JUMP to .XX will occur only if the value of the symbol INC is greater than ten.

The DO pseudo-operation is used to control an assembly time loop and may be written in one of three forms:

(*i*) DO .LOC,VAR=INIT,TEXP,INC
(*ii*) DO .LOC,VAR=INIT,LIMIT,INC
(*iii*) DO .LOC,VAR=(LIST)

Types *i* and *ii* assign the value of INIT to the variable symbol VAR and then assemble all the lines up to and including the line with .LOC in its location field. The value of INC (if INC is omitted, 1 is assumed) is added to the value of VAR. For type *i*, the truth value expression TEXP is evaluated; if it is true, the loop is repeated. Type *ii* compares the value of VAR with the value of LIMIT; the loop is repeated if INC is positive and the value of VAR is less than or equal to the value of LIMIT. If INC is negative, the loop is repeated only when the value of VAR is greater than or equal to the value of LIMIT. Type *iii* assigns to VAR the value of the first item in LIST. Succeeding values are used for each successive time around the loop until LIST is exhausted.

The following is an example of the use of DO

```
        DO      .AA,INC=1,3
        GT        INC
.AA     TCS       TBL+INC
```

The assembler will produce the same output from these three lines as from the following sequence:

```
GT    1
TCS   TBL+1
```

```
GT    2
TCS   TBL+2
GT    3
TCS   TBL+3
```

2.3 *Symbols and Attributes*

Symbolic names are limited to fewer than 250 alpha-numeric characters at least one of which must be alphabetic. The alphabetic characters are considered to include the 26 upper and 26 lower case letters as well as the underscore and percent sign. A symbol may be used to identify a program store location, call store address, or a program parameter. The value of a symbol may be a 32-bit integer, a character string, or another symbol. In addition, every symbol may have up to 250 attributes which are 24-bit integers. The x attribute of the symbol A is represented by X(A). The following line, for example, sets the ALPHA attribute of the symbol SA to ten greater than the BETA attribute of symbol SB.

```
ALPHA(SA)  SET  BETA(SB)+10
```

2.3.1 *Symbol Types*

Each symbol has an associated type character; a program store location symbol, for example, has type L. The type of each operand in an arithmetic or logical expression is used to determine the correct method of evaluating the expression. It is also used to check for illegal combinations of operands and to appropriately mark the error.

2.3.2 *Available Type Characters*

Some of the available type characters and their meanings are:

A—absolute symbol
C—call store location symbol
D—program store data location symbol
J—truth valued symbol
L—program store location symbol
N—integer
T—text symbol
X—external symbol

2.4 *Arithmetic and Logical Expressions*

Arithmetic or logical expressions consist of a string of operands separated by operators or parentheses. An operand may be an integer, symbol, function call, attribute reference, indirect symbol, or character

string. A character string in an expression is represented by enclosing the string in either single or double quotation marks and is converted to a 32-bit binary integer when used in any operation except a comparison. An indirect symbol is a symbol defined by the text pseudo-operation where the character string definition is a valid arithmetic or logical expression.

The following is a list of the available arithmetic, logical, comparison, and special operators, listed in the order of hierarchal preference; the first to be evaluated are at the top of the list. The order of evaluation may be controlled by the use of parentheses.

Special Operators

" or "	Indicates beginning or end of character string.
?	Result is true when preceded by an operand.

Arithmetic Operators

** or ↑	Exponentiation
*	Multiplication
/	Division
unary −	Negation
unary +	No operation
+	Addition
−	Subtraction

Comparison Operators

These operators are all of the same hierarchal value and yield a result of either true or false.

=	Equals
>	Greater than
<	Less than
¬= or ≠	Not equal
=>, >=, or ≧	Greater than or equal
=<, <=, or ≦	Less than or equal

The hierarchy of comparison operators is slightly different when they are used in a double relation; for example, A<B<C will have a result of true only when A is less than B and B is less than C.

Logical Operators

&	Logical intersection.
¬	The intersection of the left operand with the complement of the right operand.

unary ¬	Complement
|	Logical union
!	Logical exclusive OR

2.4.1 *Predefined Arithmetic and Logical Functions*

Several built-in or predefined functions are available to aid in evaluating some of the more common or complicated expressions. The following is a partial list of the available predefined functions:

D(EXP)	Results in the value of the expression EXP with any integers that occur interpreted as decimal.
E(EXP)	The result is 2 raised to the EXP power.
MAX(EXP_1 , · · · , EXP_n)	Results in the maximum of EXP_1 through EXP_n .
PAR(EXP)	Returns the even parity of the value of the expression EXP.
STYP(EXP,C)	Returns the value of EXP, but the type of the result is the character C.

2.4.2 *Programmer-Defined functions*

To allow the programmer to define any number of new functions, the DFN pseudo-operation is provided. The general form of a function definition is written:

$$\text{DFN } F(P_1 , P_2 , \cdots , P_n) = A_1 : B_1 , A_2 : B_2 , \cdots , A_n : B_n$$

where F is the function name, the P_k are dummy parameter names, and the A_k and B_k are any valid expressions that may contain the P_k and other variables.

To evaluate the function, the B_k are evaluated left to right. The result is the value of the A_k corresponding to the first B_k that has a nonzero or true value. If B_n is not present, it is assumed to be true; also if all the B_k are false, the value returned is zero. The parameter expressions are evaluated, and these values are used whenever a dummy parameter is encountered in the defining expression.

Two features are provided to allow an arbitrary number of arguments in the call of a function. The first is the ability to ask if an argument was implicitly omitted from the call (explicitly omitted arguments are treated as zero). This feature is invoked by a question mark immediately following the dummy parameter. If the argument was present, the result of the parameter-question mark is the value true; otherwise, the value is false. For example, the definition:

$$\text{DFN INC(X, Y)} = X + Y : Y?, X + 1$$

would yield the value 7 when called by INC(3,4) but the value of INC(3) is 4.

The other feature is the ability to loop over a part of the defining expression, using successive argument values wherever the last dummy parameter appears in the range of the loop. This feature is invoked by the appearance of an ellipsis (. . .) in the defining expression. The range of the loop is from the operator immediately preceding the ellipsis backward to the first occurrence of the same operator at the same level of parentheses. As an example, consider the following statement:

```
DFN SUM(A,B,X,Y)=A+X**(Y+1)+. . .+B/2
```

The range of the loop is from the + between the A and the X to the + following the right parentheses. The call SUM(2,18,3,1,2,3) would yield the same result as the following expression:

$$2+3^{**}(1+1)+3^{**}(2+1)+3^{**}(3+1)+18/2.$$

The loop may also extend over the expression between two commas as the next example shows. A function to do the exclusive OR of an indefinite number of arguments could be defined by:

```
DFN XOR(A,B,C)=A¬B | B¬A:¬C?,XOR(XOR(A,B),C, . . .)
```

2.5 *Macro Definitions*

The real power of an assembler lies in the flexibility it provides the programmer. The *macro* facilities incorporated in SWAP have more than the necessary features to make it one of the most powerful assemblers available. A macro instruction is an abbreviated form for a sequence of predefined instructions, pseudo-operations, or comments. Whenever a macro is called, the predefined sequence is generated in place of the macro call. The sequence of statements generated by a macro may be varied by the use of any of the several conditional assembly facilities provided.

The general form of a macro definition is:

```
MACRO
prototype statement
macro text lines
MEND
```

The prototype statement contains the name of the macro definition as well as the dummy parameter names which are used in the definition.

The macro text lines, a series of statements which make up the definition of the macro, will be reproduced whenever the macro is called.

The following is an example of a simple macro that may be used to move data from one call store location to another:

```
      MACRO
LOC MOVE FROM,TO
LOC RED FROM
      WRI TO
      MEND
```

The subsequent macro call:

```
SAVE MOVE NEWDATA,OLDDATA
```

would generate the following instructions:

```
SAVE RED NEWDATA
     WRI OLDDATA
```

2.5.1 *Macro Arguments and Operators*

The general forms of a macro argument are OP(ST) or PAR. OP is called the operation part of the argument and includes all characters up to the first left parenthesis. ST is called the strip and (ST), the body of the argument. PAR is an argument that is not of the op-strip form. Several macro operators are available to allow the programmer to obtain these parts as well as other pertinent information about an argument. A macro operator is indicated by its name character followed by a period and the dummy parameter name of the operand. For example, the operation part of a parameter named ARG is obtained by the use of O.ARG, and the strip is represented by S.ARG. Whenever the op part of a argument is requested and the argument is not of the op-strip form, a null value is returned; the strip of a non op-strip argument is the entire argument.

2.5.2 *Macro Subarguments*

Many times the strip of a macro argument consists of a sublist of parameters. Any subparameter may be accessed by subscribing the parameter name with the number of the desired subargument. Additional levels of subarguments are obtained with the use of multiple indexes. As an example, let parameter ARG assume the value P(Q,R(S,T)), then: ARG(0) represents P; ARG(1),Q; ARG(2),R(S,T); and ARG(2,2) would be replaced by T.

The macro operators may be used on the results of each other as well as on subparameters; for example, S.ARG(2) would refer to S,T.

The subargument indexes may be symbolic expressions that contain other macro parameters as the following example of a macro with a DO loop demonstrates. A macro to copy data from one call store location to any number of other call store locations could be written:

```
        MACRO
        COPY    PARM
        RED     0.PARM
        DO      .LOOP,K=1,N.PARM
.LOOP   WRI     PARM(K)
        MEND
```

The number macro operator, N, is replaced by the number of subarguments in its operand so that the DO will loop as many times as there are subarguments in PARM. The macro call:

```
COPY    DATA(SAVE,HOLD)
```

will then generate the following instructions:

```
RED     DATA
WRI     SAVE
WRI     HOLD
```

2.5.3 *Macro Functions*

To provide more flexibility with the use of macros, several system parameters and macro functions have been made available. A macro function call is replaced by the string of characters that is its result. The arguments of a macro function may consist of macro parameters, other macro function calls, literal character strings, or symbolic variables. An example would be the DEC macro function, which has a single argument that is a valid arithmetic or logical expression. The result is the decimal number equal to the value of the expression; the call DEC(7+8) would be replaced by 15.

The three major macro functions are:

(*i*) IS(*expression,string*) is replaced by *string* if the value of *expression* is nonzero; otherwise, the result is the null string.

(*ii*) IFNOT(*string*) is replaced by *string* if the *expression* in the previous IS had a value of zero; otherwise, the result is null.

(*iii*) STR(exp_1,exp_2,*string*) is replaced by exp_2 characters starting with the exp_1 character of *string*.

A more sophisticated example of the use of macro functions is this version of the COPY macro:

```
        MACRO
        COPY    PARM
        RED     0.PARM
        DO      .LOOP,K=1,N.PARM
.LOOP   1S('PARM(K)'='HOLD',HGR)1FNOT(WRI PARM(K))
        MEND
```

Using the above definition, the call:

```
        COPY    DATA(SAVE,HOLD,LOC2)
```

would expand to:

```
        RED     DATA
        WRI     SAVE
        HGR
        WRI     LOC2
```

2.5.4 *Keyword Arguments*

It is often convenient to be able to override the positional relationship hetween the dummy parameters on the macro prototype line and the arguments on a macro call. This may be done when the macro is called by writing the parameter name followed hy an equal sign and the argument string. An argument of this form is called a keyword argument. An example would be the following calls of the MOVE macro.

```
        MOVE    FROM=NEWDATA,TO=OLDDATA
```

or

```
        MOVE    TO=OLDDATA,FROM=NEWDATA
```

Both calls will expand to the same instructions as the expansion of the MOVE macro without keyword arguments.

2.5.5 *Default Arguments*

Another convenience is the ahility to have a standard, or default, value for a parameter. The default value would he used whenever the argument was omitted from the call. The default value must he assigned on the macro prototype line by an equal sign and the default value after the dummy parameter name. Another version of the MOVE macro is an example of assigning default values.

```
MACRO
MOVE    FROM=TEMP,TO=TEMP+14
RED     FROM
WRI     TO
MEND
```

The call:

```
MOVE    TO=OLDDATA
```

would then expand to:

```
RED     TEMP
WRI     OLDDATA
```

2.6 *Automatic Instruction Insertion*

No. 2 ESS instructions are put in the format of one or two per word. The half-word instructions of the No. 2 ESS occasionally cause a no-operation (NOP) instruction to be required. When there are an odd number of half-word instructions between full-word instructions or when the destination of a transfer would otherwise be in the middle of a word, a NOP is inserted by the assembler.

The half-word transfer commands specify a five bit address. The destination of such a transfer is thus limited to the same block of 32 words as the transfer. An instruction (FIL) is available to extend the addressing range to 1024 words by leaving five bits in a buffer. The assembler will give an error message each time a short transfer is used that: (*i*) requires but does not have an associated FIL, (*ii*) has a FIL that was not needed, or (*iii*) is insufficient even when a FIL is used.

When the programmer adds or deletes an instruction, a short transfer may require a FIL where it was not previously needed. SWAP inserts the appropriate FIL instruction wherever it is needed and attempts to place it where a NOP was required in the right half of a word. It is extremely difficult to have only the minimum number of FIL's and, therefore, some extra FIL commands will be inserted in the program. The automatic FIL insertion feature may be turned off if the programmer so desires.

2.7 *Text Manipulating Facilities*

Some of the more exotic features provided by the switching assembler program are the character string pseudo-operations and the dollar functions, so called because the function names all start with a dollar sign.

2.7.1 HUNT *and* SCAN *Pseudo-Operations*

The HUNT pseudo-operation allows the programmer to scan a string of characters for any break character in a second string. It will then define two TEXT symbols consisting of the portions of the string before and after the break character.

The SCAN pseudo-operation provides the extensive pattern matching facilities of SNOBOL3 along with success or failure transfer of control.[1] These features, too diverse to be discussed here, are covered in the references.

2.7.2 *Dollar Functions*

Dollar functions are very similar to macro functions in that the result of a dollar function call is a string of characters that replace the call. The dollar functions may be used on input lines as well as in macros. For example, $(TSYM) would be replaced by the character string which is the value of the text symbol TSYM. A very useful feature of the dollor functions is in the ability to call a one-line macro anywhere on a line by preceding the macro name with a dollar sign and following it with the argument list in parenthesis. For example, the macro:

```
        MACRO
        CHECK   A,B

IS(A<B,DEC(B-A) MORE)IFNOT(DEC(B-A)OVER)

        MEND
```

could be called by:

```
X       SET     5
        LGR     X   #   $CHECK(X,8)
```

but the line would appear in the assembly listing as:

```
        LGR     X   #   3 MORE
```

2.8 *The Assembly Listing*

Since the input line format for SWAP is free field, the assembly listing of the source lines may appear quite unreadable. Therefore, the normal procedure is to have the assembler align all the fields when a line is printed. For example, a programmer may punch his cards:

```
        TRA         LOC # GO BACK
REST        LGR 7;GRXAA # LOAD AA
```

but the assembly listing would show the lines thus:

```
        TRA     LOC     # GO BACK
REST    LGR     7
        GRXAA           # LOAD AA
```

The position of the fields as well as the position of the line is a programmer option. Some of the other options that are available to control the format of the listing are: double spacing, titling at the top and bottom of each page, and several classes of lines that may be printed at the programmer's discretion.

2.9 *Inputs*

SWAP may receive its original input from a card, disk, or tape data set. The SOURCE pseudo-operation allows the programmer to change the input source at any point within a program. Another source of input is the EDITOR program, which provides extensive facilities for making changes or corrections in a program.

2.9.1 *The* EDITOR *Program*

Any SWAP input line that contains a colon in column one is assumed to be an EDITOR control card and, therefore, invokes the EDITOR program. The EDITOR is then responsible for retrieving a source data set, making the indicated changes, and passing each line back to SWAP to be assembled. The EDITOR provides facilities for inserting, replacing, or removing lines as well as modifying a part of a line and moving or copying a group of lines to another position within the data set. Since the normal output of the EDITOR goes directly to the assembler, the original data set is not changed unless the programmer explicitly requests that the changes be permanently incorporated in a new copy of the data set.

2.9.2 *Libraries*

SWAP also has facilities to save symbol, instruction, or macro definitions in the form of libraries which may be loaded later in another assembly. When, for example several programs make use of a common set of macros, it is desirable to obtain them from the same source. The SOURCE pseudo-operation could be used for this, but it would require that each symbol, instruction, or macro be completely processed by the assembler. As this is relatively slow and inefficient, a method of producing a library which contains the processed definitions is provided. Later, if a program requires it, those definitions may be loaded and used, bypassing the costly definition process.

2.10 *Outputs*

The output of the assembler normally consists of the assembly listing and the object program module (see below) but other outputs may be obtained upon request. The libraries are one example. Special outputs can be produced from stylized comments included in each No. 2 ESS program. One type of comment provides information so that a flowchart of the program can be generated by machine. Another type of comment is used to produce a manual explaining all the teletypewriter messages that the program might issue.

Also provided as an optional output of the assembler are the results of the macro expansions and dollar substitutions. The programmer controls the format of each line as well as the deletion of undesired lines. This allows the assembler macro facilities to be used to produce an input data set for any of the No. 2 service programs or IBM 360 support programs.

III. LOADER

The LOADER program accepts output from SWAP assemblies. The assembled programs are combined and placed on disk storage as a paged image of the program store. All interprogram linkage is performed; that is, all external symbolic references are resolved. The program store image is in the form required by the simulator or twistor card preparation program.

3.1 *Object Program Module*

The output of a SWAP assembly is called an object program module. All object program module's of a project are normally contained in a single partitioned data set. They may exist only on a direct access device, although they may be saved on magnetic tape. There may be a number of object program module's for any program, reflecting various stages of development. The assembler creates a PRIVATE, or working copy. A utility program creates a PUBLIC copy from a relatively debugged PRIVATE copy (and pushes down the otber PUBLIC copies). The LOADER normally loads the most recent PUBLIC copies but may load others as described later under the LOAD verb.

3.2 *Outputs of* LOADER

3.2.1 *Program Store Image*

The primary output of the LOADER is a paged image of the program store. It is made up of a 2048-byte record for each two planes (512

ESS words) loaded, plus a directory to indicate which plane is on which record.

3.2.2 *Printed Output of* LOADER

The LOADER produces four printouts. These are shown in the Appendix.

All control cards processed by the LOADER are listed. Diagnostics are self-explanatory; any error flags are explained at the end of the control card listings.

Unless suppressed, a loading map is generated. This map includes, for each program loaded, the version (PUBLIC or PRIVATE), time and date of assembly, first and last locations in the program, the number of a tape (if any) on which the assembly listing is stored, and a remark. The remark indicates whether the program was implicitly or explicitly loaded (see Section 3.3). If a program could not be found in the object program module data set, it is marked undefined. Printed below the loading map is a statement of whether any areas of the program store were loaded more than once. If overwrites were present, they are listed.

If a cross-reference table is requested, it lists all external references that were resolved by the loader. This may be a very large list. If the listing is not requested, only those references for which a diagnostic is generated are listed. The cross-reference table may be sorted by symbol name.

3.3 *Implicit Loading*

Unless EXPLICIT loading is specified, IMPLICIT loading is assumed. This means that any program that is referred to by one that has been previously loaded is also loaded. It is, therefore, possible to load all programs by explicitly mentioning only one. Programs not wanted may be excluded from the loading.

3.4 LOADER *Features*

The function of the LOADER is best described by describing some of the primary verbs.

3.4.1 LOAD *Statement*

The LOAD statement describes which programs, and what versions, are to be loaded. An unqualified program name indicates the latest PUBLIC version. Program names may be qualified by PRIVATE or a date. When the PRIVATE qualification is used, the PRIVATE version is loaded. When a date is specified, the latest version assembled not later than that date is loaded.

Example: LOAD BLMM IOMAINT(PRIVATE) IOTEST(6/12/69)

3.4.2 EXCLUDE *Statement*

The EXCLUDE statement lists programs to be excluded from implicit loading.

Example: EXCLUDE CSUB PCMAINT

3.4.3 SET *Statement*

The SET statement defines or changes the value of a PUBLIC symbol; that is, a symbol to which an external reference is made. In Example 1, the symbol RETURN1 in the program BLMM is given the value 10436 octal. Example 2 equates the symbol SUB23 in the program IOSUB to the value of SUBA2 in CSUB.

Example 1: SET BLMM.RETURN1 = 10436

Example 2: SET IOSUB.SUB23 = CSUB.SUBA2

IV. SIMULATOR

The No. 2 ESS simulator, SMILE,* provides a powerful program checkout, or debugging, facility. It is also a vehicle for investigating the effects of proposed system changes.

4.1 *Simulation*

A typical user assemhles his program, loads it along with related programs, and simulates it to find errors (see Fig. 1). He requests printouts of pertinent data at points where he expects specific results. If the results are not right, he examines all of the outputs. If he fails to find the trouble, he will simulate again, producing more output around the problem area. In this way, he can close in on the error. A map of the path taken by the program at each hranch and a printout of the contents of the registers are tools for finding out what went wrong.

Proposed system changes can he evaluated using the simulator. For example, the effect on system capacity of changes in command timing can be studied by incorporating the changes into the simulator and observing the effects by simulating the call processing programs.

---

* SMILE is an acronym for switching machine interpreter for lazy engineers!

4.2 *Objectives of* SMILE

The main objectives in the design of SMILE were completeness of simulation, ease of use, and speed. It is thus possible to simulate a complete call using few control statements in a reasonable amount of IBM 360 time.

4.2.1 *Coverage*

SMILE simulates most processor commands, some No. 2 ESS input-output, but no internal wired logic. All commands used by tbe non-maintenance programmer are simulated. To perform No. 2 ESS input-output, special control statements bave been developed. Tbese enable one to place digits into originating registers at specified times. Other more general control statements allow locations in call store to be changed based on tbe reacbing of specified program store locations or the passing of a specified amount of No. 2 ESS time. Special No. 2 ESS output messages are produced on certain external commands. All of tbese features are discussed in more detail in the following paragraphs.

4.2.2 *Ease of Use*

In order to make SMILE easy to use, default values on everything possible bave been set to the most common value. For example, call store words are initially zero, and all scanners are initially ones. Each input to SMILE is written in the natural language for that item. Time, for example, may be written in cycles, microseconds, or milliseconds. Program store addresses and call store addresses are assembled and specified symbolically.

4.2.3 *Speed*

To be useful in a practical environment, a simulator must be fast. The simulation of a typical ESS instruction takes less than 30 microseconds on an IBM 360 model 65. This basic simulation ratio of better than 10 to 1 (elapsed time to simulated ESS time) is achieved by trading space for speed in the more common routines.

Anotber contributor to SMILE's speed is tbe preprocessing of control statements tbat will be performed during simulation. Control statements are converted to interpretive code which is executed each time the function is performed. The interpretive code produced must run fast, witb little consideration given to bow long it takes to produce tbe code. The interpretive code produced must not use a lot of core

because this would greatly limit the number of control statements possible.

Generation of outputs consumes a large portion of running time, so SMILE limits unnecessary printing. Unless told otherwise, SMILE only prints a final dump of the nonzero registers and call store, and a few error and special messages. Error messages are automatically turned off after ten of a kind have been printed. To produce more output, the user must explicitly ask for it.

4.3 *Control Language*

4.3.1 *Initializing Statements*

The control language that has been developed is natural and easy to use. To initialize a register, one merely writes an "=" statement as follows: (*i*) name of register, (*ii*) =, and (*iii*) value to be stored in the register. For example, GR = 1243. Since No. 2 ESS programmers are accustomed to octal code, data constants are interpreted as octal numbers. Counts and time are interpreted as decimal numbers. To initialize a word in call store, one writes an "=" statement as follows: (*i*) symbolic name of area in call store or octal address, (*ii*) =, and (*iii*) value to be stored. For example, CSTBL = 421 or 2310 = 1337.

Of the two groups of scanners in the No. 2 ESS, one is primarily used for lines and the other for trunks. "TRUNK(3,4)" refers to row number 4 in trunk scanner number 3. "LINE(2,5,3)" refers to bit number 3 of row number 5 of line scanner number 2. These functions may be used on the left side of "=" statements to initialize the scanners. For example, TRUNK(3,7,2) = 1 or LINE(5,4) = 176777.

4.3.2 *Plants in Program Store*

Simulation can be interrupted when specified program store locations are reached. At a location, a set of features can be planted. These would be performed each time that location is reached. Initialization statements as well as other features can be "planted" at a location.

4.3.3 *Time and Automatic Interrupts*

As each command is simulated, the timer is incremented by the amount of time the command takes in the No. 2 ESS. Concurrently, a check is made to see if the user has requested an interrupt at this point. There are three ways a user can request such an interrupt:

(*i*) The TIME control statement says to perform the "range" (control

statements after the TIME and before the next TIME or plant) when the timer reaches the time specified by the first operand. If there is a second operand, the range will be performed again when that amount of time has elapsed, and each time thereafter. If a third operand exists, it tells when to stop processing this TIME.

(*ii*) The AFTER control statement is similar to the TIME control statement except that it is planted at a location. Its first and second operands correspond to the second and third operands of the TIME control statement. It enables a programmer to interrupt after a certain amount of time has passed after reaching a certain location.

(*iii*) The INOUT control statement generates other control statements which create an INOUT interrupt at the time indicated by the first operand. For ease of use, 25 milliseconds is assumed if the first operand is missing.

4.3.4 *Digit Insertion into Originating Register*

A digit control statement places digits into an originating register in call store, one for each time the control statement is encountered. The verb DIGDP, DIGMF, or DIGTT indicates how the second operand list is to be interpreted—dial pulse, multifrequency, or *Touch-Tone*®-dialing, respectively. The first operand indicates which originating register receives the digit. The second operand is a list consisting of symbolic codes indicating which digits are to be deposited.

If, as is usually the case, the digit control statement is used in the range of a TIME, the first digit is deposited at the time indicated hy the first operand of the TIME, and successive digits are deposited at increments of time indicated by the second operand of the TIME. If the digit control statement is planted in a range at a program store location, the digits are deposited one at a time each time that program store word is simulated.

4.3.5 *Symbolic Input*

SMILE has heen designed to allow symbolic reference to program store locations. A program may thus be changed without modification of the control statements. The SWAP assembler produces a symbol tahle with equivalences which the simulator uses. When symbols from several programs are referred to, it is necessary to indicate to which program each symbol belongs. The prefix notation is one way to do this. INI.LOOPMORE is used to refer to the location LOOPMORE in program INI. Prefix notation is used when a small number of symbols are needed from a given program.

The SYMBOLS PRNAME control statement informs SMILE that all nonprefixed symbols between here and the next SYMBOLS control statement are to be found in the symbol table from the program PRNAME.

4.3.6 *Conditions*

The ability to conditionally perform various control statements dependent on the comparison of variables is an essential part of the simulator control language. The IF and UNLESS verbs enable one to perform these comparisons in a natural manner.

"IF GR = 2" means to perform the next control statement if the contents of the general register (GR) is 2. Four relations are permitted: = (equal), < (less than), > (greater than), and ¬ (not—used as a prefix). Combinations are also permitted: <=, >=, ¬=, ¬>, ¬<=, and so on. The not sign (¬) negates the entire relation regardless of where it occurs.

"UNLESS LR=4" does the same thing as "IF LR ¬= 4." A condition applies to the single control statement that follows it, unless that statement is a BEGIN. Then the condition applies to the "block" of control statements starting with the BEGIN and ending with a paired END. Blocks may be nested up to a maximum depth of ten.

The expressions used on both sides of relations and "=" statements may include registers, constants, call and program store symbols, scanner functions, and the special dummy registers X0 through X20 combined with the following operators:

@	indirection (constant refers to call store)
**	integer exponentiation
*	multiplication
/	integer division (truncated)
+	addition
−	subtraction
&	logical AND
|	logical OR
()	parentheses may be used to alter the above hierarchy (operations on top are performed first).

For an example, consider: "IF GR = @(CSTBL+X0)*2**5&3740." The contents of dummy register X0 (which was previously set by something like X0 = 12 or X0 = X0+1) is first added to the address CSTBL. This sum is used to index into call store. The word thus obtained is saved while 2 is raised to the fifth power. Next, the saved word is multiplied by the power and the result is masked by 3740.

For another example, consider: "IF AA = LR|GR&AA –CA* LW**@CSEXP." The operations will he performed in the exact reverse order of the way they are written since eacb successive operator is higher in the ahove list tban its predecessor.

4.3.7 *Normal Sequence Breakers*

In the process of using conditions, it is often desirable to jump over a set of control statements. This can be done by using the JUMP verb followed hy a sequence symbol and by placing tbat sequence symhol just hefore the control statements with which processing should continue. This is a branch among control statements. Sequence symbols are by definition symbols whicb start witb a period. For example, in the following input stream: ".CASE1" and ".CASE2" are sequence symbols:

```
IF GR = 1236 JUMP   .CASE1
IF LR = 42   JUMP   .CASE2
JUMP   .OUT
.CASE1   GR = 12   SNAPTR   JUMP   .OUT
.CASE2   LR = 1232 SNAPTR   JUMP   .OUT
```

"JUMP .OUT" naturally means to jump out of this range and go back to simulation at the point it was interrupted.

It is often desirahle to skip over a section of program or go to some other place. This can be done hy using the GOTO verb followed by a program store address (symholic or constant). For example, GOTO LOOPMORE+2, or GOTO LOOPMORE–1. This is a brancb from performing control statements to the simulation of a particular program store word.

The verb whicb specifies the end of control statement processing and the start of simulation is START followed by tbe program store address wbere simulation is to begin. No address means simulation starts at the origin of the first program loaded.

4.4 *Outputs*

4.4.1 *Transfer Trace*

Transfer tracing is the process of following the flow of a program hy printing out a line every time an instruction is executed out of sequence. Each line of trace output includes the type of branch and ten data words. The different types of branches are:

ADV	advance command (when progress mark found)
GOTO	GOTO control statement

PA=	= control statement
PIBn	program interrupt begin command
PIEn	program interrupt end command
TRACE	all transfer commands (when transfer is taken).

There are two control statements used with tracing:

```
TRACE N
FORMAT LIST.
```

The TRACE N control statement starts tracing and continues until N lines have been printed. It may be placed in the initial input stream, at a location, or in the range of a TIME. In the FORMAT control statement, LIST indicates which registers, call and program store words, scanner rows, and special registers X0–X20 are to be printed. Furthermore, the FORMAT statement is dynamic; it can be used not only in the initial input stream but also as a plant or in the range of a TIME. If a FORMAT statement is not encountered first, a default FORMAT is automatically generated whose LIST is "GR, CA, @CA, AA, KM, LR, LM, LW, RF, TB."

4.4.2 DSNAP *and* SNAPTR

A handy debugging device is the DSNAP statement which prints out all registers and call store words that have changed since the last DSNAP. The first DSNAP prints out all nonzero words. This type of output is slow since it has to compare the current contents of all of call store with the previous contents and also save the current as the new previous and print several lines showing changes. This kind of tool, although slow, can find errors that would otherwise go undetected. A wild write into call store shows up very quickly.

Whenever it is encountered, the SNAPTR feature causes the printing of a TRACE line. It and all the other features can be planted at a program store location or caused to happen at a given time.

The DATATR feature is a special feature which causes the printing of a TRACE line after each occurrence of the DATA command. The DATA command is used to retrieve data, such as translations, from the program store. A DATA trace effectively records the progress of a call by monitoring the translators. DATATR is usually placed in the initial input stream but can also be planted or put in the range of a TIME.

4.4.3 *Special Messages*

Most of the external commands are accompanied by special printouts. These printouts give pertinent information so the programmer can

see if the program produces the necessary outputs to perform the desired switching. Several other special messages are associated with the hold-get counter and transfers.

PRINT control statements control the printing of the above messages. The default case is to print each message ten times and then suppress the message. The PRINT control statement can turn a message OFF, or ON, or ON for a certain number of times. This statement is also dynamic—the ON-OFF-COUNT status of any message can be changed during simulation.

4.4.4 *Final Dump*

At the completion of each simulation, an automatic dump of all nonzero registers and nonzero call store locations is given. The dump takes the form of a DSNAP assuming the previous DSNAP was all zeros.

4.4.5 *Symbolic References to Program Store*

With each request for output, the following header prints on the left side of each line:

(*i*) The current No. 2 ESS time.

(*ii*) The current contents of the program address register. (The location of the plant. On transfers, the from location.)

(*iii*) The symbolic equivalent of the program address which includes the symbolic name of the program, the nearest previous symbol, and tbe increment from the symbol.

(*iv*) The name of the feature.

On features requiring multiple line output (like DSNAP), the beader information appears only on the first line. The remaining fields are a function of the feature requested. For example, while tracing, there are ten data fields. For special messages, it may be a sentence with a variable inserted. Examples are shown in the appendix.

4.4.6 *Teletypewriter Output*

During simulation, a user's program may request the printing of No. 2 ESS teletypewriter messages. This is done by having the No. 2 ESS teletypewriter program produce printing and control characters, one at a time, during the 25 millisecond interrupt. This stream of characters is saved on a scratch data set. When the simulation of the user's program is complete, control may be passed to a special set of control statements which will cause the teletypewriter program, via a fast simulation, to generate all characters in any remaining messages.

When all simulation is complete, the teletypewriter post processor sorts by teletypewriter number and prints all tbe saved characters. This last step is omitted if the user never starts any teletypewriter messages.

4.5 *Internal Structure*

SMILE consists of three basic divisions: input control statement interpreter, command simulator, and teletypewriter message processor.

During the first pbase, all control statements are scrutinized. Flags are planted where indicated functions are to be accomplished, the interpretive code required to evaluate expressions is generated, and a queue of all time-based interrupts is formed.

When the last control statement, START, is encountered, pbase two, the actual simulation, commences. Each command from the program is simulated hy a subroutine selected by using its operation code as an index into a transfer table. Input and output functions are performed when flags requesting them are encountered. The interrupt list is checked each time a command subroutine increases the timer. Ranges are executed whenever they are reached. Expressions are dynamically evaluated each time they are encountered and conditions are performed based on these results. Whenever a "STOP" or an unrecoverable error occurs, the simulation phase stops, and post processing takes place. Any teletypewriter messages generated during simulation are finally printed.

V. SUMMARY

The service programs described in this article comprise ahout 50,000 words, about two-thirds of whicb are the assembler. These programs are now in use for the No. 2 ESS project. Tbe assembler was designed to be common to several projects at Bell Laboratories, and the authors wish to acknowledge the contribution of Messrs. R. E. Archer, A. J. Emrick, and E. Walton to the design and implementation of this program.

Typical execution times are one minute for assemblies (up to two minutes for 250 page assemhlies), one minute to load all programs, and one to three minutes for simulations. The use of the powerful facilities of SWAP varies greatly among programmers. Some make extensive use of the macro features; others, almost no use. This appears to be caused by the difficulty of learning these features well enough to use them effectively. Once a programmer overcomes this obstacle, he uses the tools often.

Most system programs have heen extensively simulated; as a result,

they were relatively debugged before being tested in the laboratory model. For example, only five program bugs were found using the laboratory model before the first call was completed. The simulator is expected to continue to be a valuable tool for the life of No. 2 ESS program development.

REFERENCES

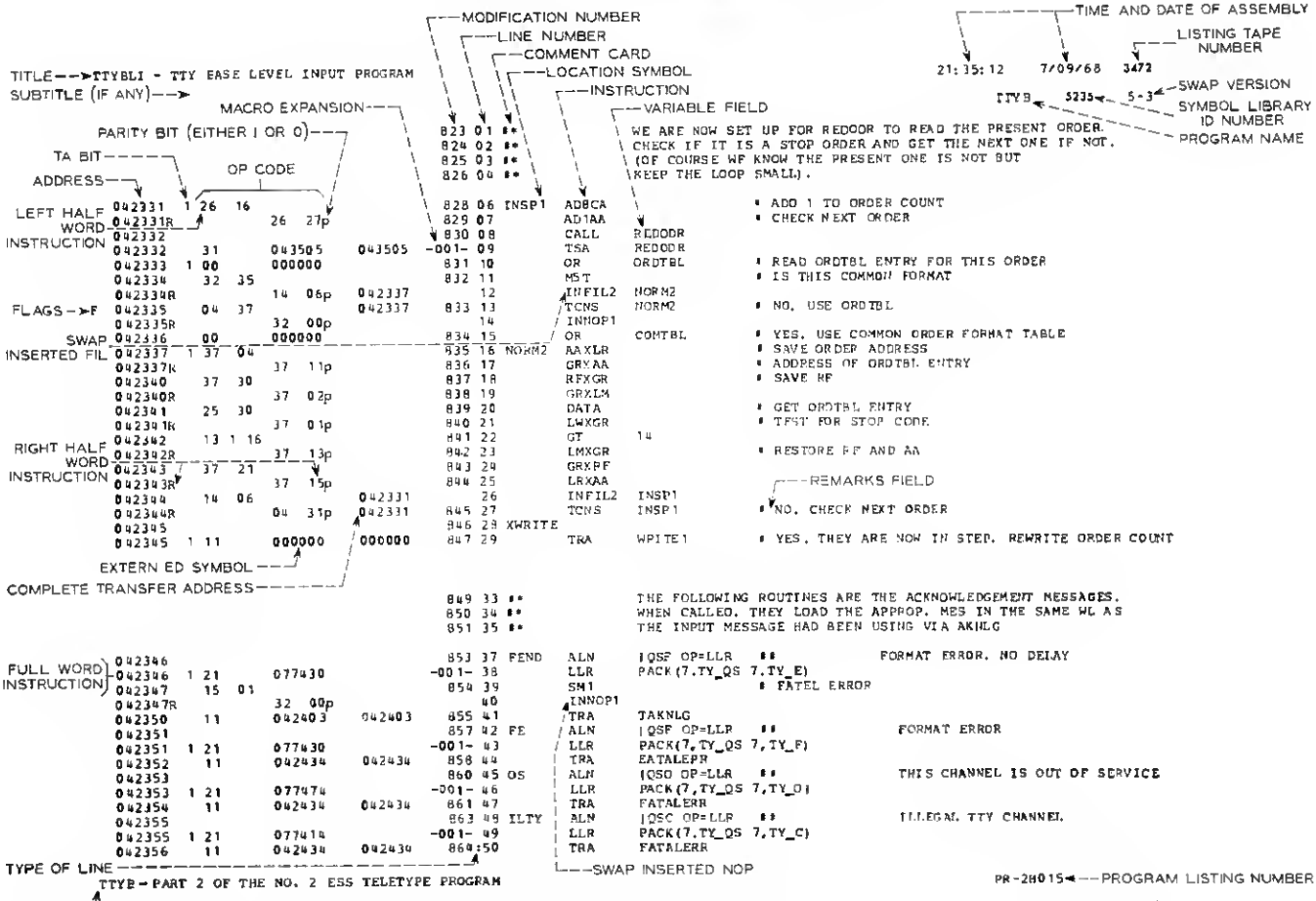
1. Farber, D. J., Griswold, R. E., and Polonsky, I. P., "SNOBOL, A String Manipulation Language," J. of the Association for Computing Machinery, *11*, No. 1 (January 1964), pp. 21–30, and "The SNOBOL3 Programming Language," B.S.T.J., *45*, No. 6 (July 1966), pp. 895–944.

APPENDIX

Computer Samples

The tables on the following pages are photographic reproductions of computer printouts.

TABLE I—SAMPLE ASSEMBLY LISTING




```
042331  1 26  16                    823 01 *#
042331R                 26  27p      824 02 *#
042332                               825 03 *#
042332          31      043505  043505 826 04 *#
042333  1 00            000000       827 05
042334          32  35               828 06 INSP1  ADBCA   INFIL2  * ADD 1 TO ORDER COUNT
042334R                 14  06p  042337 829 07       ADIAA   NORM2   * CHECK NEXT ORDER
042335          04  37       042337 830 08       CALL    REDODR
042335R                 32  00p      831 10       TSA     REDODR
042336          00      000000       832 11       OR      ORDTBL      * READ ORDTBL ENTRY FOR THIS ORDER
042337  1 37    04                   833 12       MST     CONTBL      * IS THIS COMMON FORMAT
042340          37  30               834 13       INFIL2  NORM2
042340R                 37  02p      835 14       TCNS    NORM2       * NO, USE ORDTBL
042341          25  30               836 15       INNOP1
042341R                 37  01p      837 16 NORM2 OR      CONTBL      * YES, USE COMMON ORDER FORMAT TABLE
042342  13 1 16                      838 17       AAXLR               * SAVE ORDER ADDRESS
042342R                 37  13p      839 18       GRXAA               * ADDRESS OF ORDTBL ENTRY
042343  37  21                       840 19       RFXGR               * SAVE RF
042343R                 37  15p      841 20       GRXLM
042344  14  06      042331           842 21       DATA                * GET ORDTBL ENTRY
042344R                 04  31p  042331 843 22       LWXGR               * TEST FOR STOP CODE
042345                               844 23       GT      14
042345  1 11            000000  000000 845 24       LMXGR               * RESTORE RF AND AA
                                     846 25       GRXRF
                                     847 26       LRXAA
                                     848 27       INFIL2  INSP1       * NO, CHECK NEXT ORDER
                                     849 28       TCNS    INSP1
                                     850 29 XWRITE
                                     851 30       TRA     WPITE1      * YES, THEY ARE NOW IN STEP. REWRITE ORDER COUNT

                                     852 31 *#            THE FOLLOWING ROUTINES ARE THE ACKNOWLEDGEMENT MESSAGES.
                                     853 32 *#            WHEN CALLED, THEY LOAD THE APPROP. MES IN THE SAME WL AS
                                     854 33 *#            THE INPUT MESSAGE HAD BEEN USING VIA AKHLG

042346                               855 34
042346  1 21            077430       856 35       ALN     |QSF OP=LLR  **          FORMAT ERROR, NO DELAY
042347  15  01                       857 36       LLR     PACK(7,TY_QS 7,TY_E)
042347R                 32  00p      858 37       SM1                 * FATEL ERROR
042350          11      042403  042403 859 38       INNOP1
042351                               860 39       TRA     TAKNLG
042351  1 21            077430       861 40 FE    ALN     |QSF OP=LLR  **          FORMAT ERROR
042352          11      042434  042434 862 41       LLR     PACK(7,TY_QS 7,TY_F)
042353                               863 42       TRA     FATALERR
042353  1 21            077474       864 43 QS    ALN     |QSO OP=LLR  **          THIS CHANNEL IS OUT OF SERVICE
042354          11      042434  042434 865 44       LLR     PACK(7,TY_QS 7,TY_O)
042355                               866 45       TRA     FATALERR
042355  1 21            077414       867 46 ILTY  ALN     |QSC OP=LLR  **          ILLEGAL TTY CHANNEL
042356          11      042434  042434 868 47       LLR     PACK(7,TY_QS 7,TY_C)
                                     869 48       TRA     FATALERR
```

TABLE II—LOADER CONTROL CARDS

```
          PRINT     XREF
          LOAD      IO
          LOAD      TESTPR1(PRIVATE)
          LOAD      TESTPR2  TESTPR5
L         EXCLUDE   TESTPR4,TESTPR5   * PRGMS NOT YET DEFINED.
          *
          *              UGLY NOT AVAILABLE, USE TESTPR1
             ALIAS    TESTPR1 UGLY
             EXCLUDE UGLY
U         MISTAKE
          SET       TESTPR2.SYMI2 = 5236
          SET       PR13.ALPHA = TESTPR2.BETA
          SET       CPDIG.IOSTOP = IOMAINT.IOPAUSE
          SET       CPDIG.NSTORES = 3
          ALIAS     IO,IOMAINT
          ENDL
```

L FLAG MEANS PROGRAM TO BE EXCLUDED WAS EXPLICITLY LOADED ABOVE.
U FLAG MEANS INVALID VERB - IGNORED.

TABLE III—LOADING MAP

VERSION	PROGRAM	ORIGIN	LAST	ASSEMBLED		REMARKS	TAPE
PRIVATE	IO	17600	17664	11:33:33	2/26/69	EXPLICIT	F072
PRIVATE	TESTPR1	03002	05012	17:35:00	1/20/69	EXPLICIT	AT05
PUBLIC 0	TESTPR2	10000	15002	10:47:44	1/15/69	EXPLICIT	CB50
PUBLIC 0	TESTPR5					UNDEFINED	
PUBLIC 0	CPDIG					UNDEFINED	
PUBLIC 0	TESTPR3	20130	20145	9:14:36	12/06/68	IMPLICIT	1326

OVERWRITES WERE ENCOUNTERED. SEE FOLLOWING PAGE.

Table IV—Cross Reference Table

FLAGS	LOCATION IN PROGRAM	REFERS TO SYMBOL	IN PROGRAM	WITH VALUE	FOUND IN PROGRAM	WITH VALUE
	017614	IO	IOSTOP	CPDIG 017630		
	017600	IO	NSTORES	CPDIG 000003		
	017630	IO	NSTORES	CPDIG 000003		
	003002	TESTPR1	ALPHA	TESTPR2 010000		
	003003	TESTPR1	ALPHA	TESTPR2 010000		
	001002	TESTPR1	ALPHA	TESTPR2 010000		
	003004	TESTPR1	BETA	TESTPR2 010001		
	001003	TESTPR1	BETA	TESTPR2 010001		
	003010	TESTPR1	DELTA	TESTPR2 015000		
	003011	TESTPR1	DELTA	TESTPR2 015000		
	003012	TESTPR1	DELTA	TESTPR2 015000		
	003013	TESTPR1	DELTA	TESTPR2 015000		
	003014	TESTPR1	DELTA	TESTPR2 015000		
	003015	TESTPR1	DELTA	TESTPR2 015000		
	003016	TESTPR1	DELTA	TESTPR2 015000		
	003017	TESTPR1	DELTA	TESTPR2 015000		
	003020	TESTPR1	DELTA	TESTPR2 015000		
	003021	TESTPR1	DELTA	TESTPR2 015000		
	003022	TESTPR1	DELTA	TESTPR2 015000		
	003023	TESTPR1	DELTA	TESTPR2 015000		
	003024	TESTPR1	DELTA	TESTPR2 015000		
	003025	TESTPR1	DELTA	TESTPR2 015000		
	003026	TESTPR1	DELTA	TESTPR2 015000		
	003027	TESTPR1	DELTA	TESTPR2 015000		
	003030	TESTPR1	DELTA	TESTPR2 015000		
	003031	TESTPR1	DELTA	TESTPR2 015000		
	003032	TESTPR1	DELTA	TESTPR2 015000		
	003033	TESTPR1	DELTA	TESTPR2 015000		
	003003	TESTPR1	GAMMA	TESTPR2 010005		
	003007	TESTPR1	GAMMA	TESTPR2 010005		
	005065	TESTPR1	ZEBRA	TESTPR3 020130		
P	010002	TESTPR2	ABLE	TESTPR2	TESTPR1	005065
P	010005	TESTPR2	PS1	TESTPR2	TESTPR1	005062
P	010000	TESTPR2	PS2	TESTPR2	TESTPR1	005064
P	015000	TESTPR2	PS2	TESTPR2	TESTPR1	005064
P	020136	TESTPR3	ALPHA	UGLY	TESTPR2	010000
P	020135	TESTPR3	GAMMA	UGLY	TESTPR2	010005
	020137	TESTPR3	PS1	UGLY 005062		
	020140	TESTPR3	PS2	UGLY 005064		

END OF SECO PROCESSING. ELAPSED TIME = :10

TABLE V—SIMULATOR INPUT

```
* SIMULATOR CONTROL STATEMENTS:
                FORMAT GR,CA,@CA,AA,LINE[5,4],X0,PSYM,CSYM,LM,KM
                SYMBOLS MYPRGM
                TRACE 400
                2000 = ORIG
                INOUT 3MS
LOOP            SNAPTR
AUXPRGM.MID     SNAPTR
RET             SNAPTR
CHECK           DSNAP  GR=1243
                IF CF=1 BEGIN CF=0 IF CA<120 GOTO NEXT END
                     ELSE BEGIN CF=1 IF CA<200 GOTO AGAIN END
                IF LR¬=200 CSTBL=421
                2310=1337
TEST            X0=X0+1
                IF X0=1 BEGIN TRUNK[3,7,2]=1 CA=@100[3 GOTO TEST1 END
                IF X0=2 BEGIN LINE[5,4]=176777 GOTO TEST2 END
                IF X0>=3 STOP
MIDWAY          AFTER 6US GOTO INTPRGM.INT3
TIME 14MS,10MS DIGDP ORID,6,8,2,2,2,7,9
TIME 50MS       STOP
START           MYPRGM.BGN
```

TABLE VI—SIMULATOR OUTPUT

```
  TIME    SYMBOL ICC       CCT LOC  VERE   DATA                                                          PAGE     1

                                    FORMAT   GR       CA       BCA      AA       RM       LR       IM       LW       RF       TE

0000000  OPSVC.HPTT1       101656   START

0000006        PPTT1+1     101657   TRACE    101038   004000   102443   000000   000000   000000   000000   000000   000000   000000

0000076        CRDPTR+6    101342   DATA     167001   004006   010014   101037   000000   000000   000077   013670   000006   000000

0000076        CROPTR+1    101343   TRACE    167001   004006   010014   101037   000000   000000   000077   013670   000006   000000
0000120        BPTT1+6     101664   TRACE    032002   004001   032002   101037   000000   156013   000077   013670   000003   000400
0000159  IO25M.PIC1S+8     136261   TRACE    032002   004001   032002   156113   000000   156013   000077   013670   000003   000400
0000207        ALTENT+6    136226   TRACE    000000   004001   032002   156115   000000   106400   000006   000002   000002   000400
0000231        TRCUEL+3    136244   TRACE    000000   004001   032002   156115   000000   106400   000006   000002   000000   000400
0000261        OSED+2      136242   TRACE    100000   004001   032002   156115   000000   106400   000006   000002   000000   000400

0000306        PDC7X+8     136235   MXPOC:      EA    106400       LR   156115       KR   000010       IC   000000       GR   156115

0000390        PDC7X+5     136236   TRACE    156115   004001   032002   156115   000400   115670   177777   106400   000000   000000
0000402        AB258       136240   TRACE    156115   004001   032002   156115   000000   115670   177777   106400   000000   000000

0000432    101670          101670   DSNAP-
                                           AA   156115       CA   004007      CSA   004007       EA   106400
                                           GR   101701       BG   000037       LF   000016       IM   177777
                                           IR   115670       LW   106400       IC   000220
                                    00000  000000   000240   000000   000000   000000   000000   000000   000000
                                    00360  101665   000000   000000   000000   032002   000000   000000   000000
                                    04000  102443   032002   003261   000000   000000   167001   010014   101701
                                    13200  000000   000000   000000   000000   100000   000000   000000   000000

0000432                     101670   GOTO     101701   004007   101701   156115   000000   115670   177777   106400   000000   000000

0000438  OPSVC.PPTT2+1      101702   TRACE    000000   004007   101101   156115   000000   115670   177777   106400   000000   000000
0000450        ISCAN+1      101016   TRACE    000000   004007   101101   000000   000000   115670   177777   106400   000000   000000
0000489  CSUE.CONET1+2      025615   TRACE    000000   004007   101701   000000   000000   000000   177777   106400   000006   000400
0000522        ZERO+2       025627   TRACE    000000   004007   101701   000000   000000   000000   177177   106400   000000   000400
0000543        GCC0ZE+2     025643   TRACE    070000   004007   101701   000000   000000   000000   177777   106400   000000   000400

0000555        CK+1         025617   XTSC:       EA    070000       IR   177777       IC   000000

0030586  RCSUE.             100010   A PROGRAM STORE LOCATION THAT WAS UNLOADED OR CONTAINED DATA WAS ENCOUNTERED.  STOP WAS GENERATED:

0030586                     100010   STOP.  THE LAST SIMULATED TRANSFER WAS FROM 101712.

0036586                     100010   FINAL CONTENTS OF REGISTERS AND CALL STORE:
                                           AA   003261       CA   004005      CF   000001      CSA   004005
                                           EA   070153       GR   047200      HG   000037       IF   000006
                                           LM   177403       LR   177402      RF   000007       SA   177777
                                           TC   052525
                                    00000  000000   000240   000000   000000   000000   000000   000000   000000
                                    00350  101077   000000   000000   000000   000000   000000   000000   000000
                                    00360  101764   000000   000000   000000   032002   000000   000000   000000
                                    08640  000000   070000   000000   000000   000000   000000   000000   000000
                                    04000  102443   032002   003261   003000   000000   047200   060014   101762
                                    13200  000000   000000   000000   000000   100000   000000   000000   000000

  END OF SECC PROCESSING.  ELAPSED TIME =   :37
```